



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Laboratorio de Computacion Salas A y B

Profesor(a):

Asignatura:

Grupo:

No de practica(s):

Integrante(s):

No de lista o brigada:

Semestre:

Fecha de entrega:

Observaciones:

Calificacion:

Índice

1. Introducción	2
1.1. Planteamiento del problema	2
1.2. Motivación	2
1.3. Objetivos	2
2. Marco Teórico	3
2.1. Clases y objetos	3
2.2. Constructores	4
2.3. Encapsulación, Getters y Setters	6
2.4. Herencia	8
2.5. Polimorfismo	10
2.6. Clase abstracta	11
2.7. Archivo y dart:io	13
2.8. Manejo de errores	16
2.9. Programación asíncrona: futuros, asíncrono, espera	16
2.9.1. Future	17
2.9.2. Trabajar con futuros: async y await	17
2.10. Patrones de diseño	18
2.10.1. Tipos de patrones de diseño	19
3. Desarrollo	22
3.1. Archivos	22
3.2. Hilos	25
3.3. Patrones de diseño	28
3.3.1. Implementación de patrón en una gestión de impresoras	28
3.3.2. Juego de combate Pokémon	30
4. Resultados	33
4.1. Archivos	33
4.2. Hilos	35
4.3. Patrones	37
5. Conclusiones	39
Referencias	40

1. Introducción

En el estudio de la programación no solo se requiere de la comprensión de los lenguajes y sus sintaxis, sino también la capacidad de analizar cómo los distintos conceptos se integran en el diseño y funcionamiento de un sistema. En este sentido, la práctica sobre archivos, hilos y patrones de diseño busca ofrecer una visión integral de tres componentes esenciales en el desarrollo de software, apoyándose en representaciones gráficas mediante diagramas UML estáticos y dinámicos y en la interpretación crítica de ejemplos concretos.

1.1. Planteamiento del problema

En el desarrollo de software, los estudiantes se deben enfrentar al reto de comprender y aplicar conceptos que van más allá de la programación básica. El manejo de archivos permite la persistencia y organización de datos; los hilos introducen la complejidad de la ejecución concurrente y la sincronización de procesos; mientras que los patrones de diseño ofrecen soluciones estructuradas a problemas recurrentes en la arquitectura de sistemas. Sin embargo, la dificultad radica en integrar estos elementos de manera coherente, comprendiendo no solo su implementación técnica, sino también su impacto en la calidad, eficiencia y mantenibilidad del código.

1.2. Motivación

La práctica surge de la necesidad de fortalecer las competencias en programación avanzada mediante ejemplos concretos que permitan visualizar cómo los conceptos teóricos se aplican en situaciones reales. El uso de diagramas UML estáticos y dinámicos se convierte en una herramienta clave para representar tanto la estructura como el comportamiento del sistema, facilitando la interpretación y el análisis crítico. Además, trabajar con material audiovisual proporcionado por el docente asegura un aprendizaje contextualizado, donde los estudiantes pueden relacionar la teoría con casos prácticos y desarrollar una visión más integral del diseño de software.

1.3. Objetivos

Se realizara un reporte con los siguientes temas:

- Explicación de la aplicación del tema de Archivos en el ejemplo principal del video.
- Explicación de la aplicación del tema de Hilos en los ejemplos del video.
- Explicación de la aplicación del tema de Patrones de Diseño en el ejemplo principal del video.

Con base en los códigos, se reportara lo siguiente:

- Diagramas UML estático.
- Diagramas UML dinámico.
- Interpretación de conceptos aplicados en el código.

2. Marco Teórico

2.1. Clases y objetos

Dart es un lenguaje orientado a objetos con clases y herencia basada en mixins. Cada objeto es una instancia de una clase, y todas las clases, excepto Null, descienden de Object. La herencia basada en mixins significa que, aunque cada clase (excepto la clase superior, ¿Object?) tiene exactamente una superclase, el cuerpo de una clase se puede reutilizar en múltiples jerarquías de clases. Los métodos de extensión son una forma de añadir funcionalidad a una clase sin cambiarla ni crear una subclase. Los modificadores de clase permiten controlar cómo las bibliotecas pueden subtipificar una clase.[1]

En la programación orientada a objetos, un objeto es una unidad autocontenida de código y datos. Los objetos se crean a partir de plantillas llamadas clases. Un objeto se compone de propiedades (variables) y métodos (funciones). Un objeto es una instancia de una clase.[1]

- Utilizando miembros de clase:

Los objetos tienen miembros que consisten en funciones y datos (métodos y variables de instancia, respectivamente). Al llamar a un método, se invoca sobre un objeto: el método tiene acceso a las funciones y datos de ese objeto. Utilice un punto (.) para referirse a una variable de instancia o método:

```
1 var p = Point(2, 2);
2
3 // Get the value of y.
4 assert(p.y == 2);
5
6 // Invoke distanceTo() on p.
7 double distance = p.distanceTo(Point(4, 4));
```

Utilice ?. en su lugar . para evitar una excepción cuando el operando más a la izquierda sea nulo:

```
1 // If p is non-null, set a variable equal to its y value.
2 var a = p?.y;
```

- Utilizando constructores Puedes crear un objeto usando un constructor. Los nombres de los constructores pueden ser 'init' ClassName o 'init'. Por ejemplo, el siguiente código crea objetos usando los constructores 'init' y 'init': ClassName.identifierPointPoint()Point.fromJson().[1]

```
1 var p1 = Point(2, 2);
2 var p2 = Point.fromJson({'x': 1, 'y': 2});
```

El siguiente código tiene el mismo efecto, pero utiliza la new palabra clave opcional antes del nombre del constructor:

```
1 var p1 = new Point(2, 2);
2 var p2 = new Point.fromJson({'x': 1, 'y': 2});
```

- Variables y métodos de clase

utilice la static palabra clave para implementar variables y métodos de toda la clase.

```
1 class Queue {
2     static const initialCapacity = 16;
3
4 }
5
6 void main() {
7     assert(Queue.initialCapacity == 16);
8 }
```

- métodos estáticos

Los métodos estáticos (métodos de clase) no operan sobre una instancia y, por lo tanto, no tienen acceso a ella `this`. Sin embargo, sí tienen acceso a las variables estáticas. Como muestra el siguiente ejemplo, los métodos estáticos se invocan directamente en una clase.[1]

```
1 import 'dart:math';
2
3 class Point {
4     double x, y;
5     Point(this.x, this.y);
6
7     static double distanceBetween(Point a, Point b) {
8         var dx = a.x - b.x;
9         var dy = a.y - b.y;
10        return sqrt(dx * dx + dy * dy);
11    }
12 }
13
14 void main() {
15     var a = Point(2, 2);
16     var b = Point(4, 4);
17     var distance = Point.distanceBetween(a, b);
18     assert(2.8 < distance && distance < 2.9);
19     print(distance);
20 }
```

2.2. Constructores

Los constructores son funciones especiales que crean instancias de clases.

Dart implementa muchos tipos de constructores. Excepto los constructores por defecto, estas funciones utilizan el mismo nombre que su clase.[1]

1. constructores generativos Para instanciar una clase, utilice un constructor generativo.

```
1 class Point {
2     // Instance variables to hold the coordinates of the point.
3     double x;
```

```

4   double y;
5
6   // Generative constructor with initializing formal parameters:
7   Point(this.x, this.y);
8 }

```

2. Constructores por defecto

Si no declaras un constructor, Dart utiliza el constructor predeterminado. El constructor predeterminado es un constructor generativo sin argumentos ni nombre.^[1]

3. Constructores con nombre

Utilice un constructor con nombre para implementar múltiples constructores para una clase o para proporcionar mayor claridad:

```

1  vconst double xOrigin = 0;
2  const double yOrigin = 0;
3
4  class Point {
5      final double x;
6      final double y;
7
8      // Sets the x and y instance variables
9      // before the constructor body runs.
10     Point(this.x, this.y);
11
12     // Named constructor
13     Point.origin() : x = xOrigin, y = yOrigin;
14 }

```

Una subclase no hereda el constructor con nombre de la superclase. Para crear una subclase con un constructor con nombre definido en la superclase, implemente dicho constructor en la subclase.

4. Constructores constantes

Si tu clase produce objetos que no cambian, haz que estos objetos sean constantes en tiempo de compilación. Para ello, define un `const` constructor con todas las variables de instancia establecidas como `final`.^[1]

```

1  class ImmutablePoint {
2      static const ImmutablePoint origin = ImmutablePoint(0, 0);
3
4      final double x, y;
5
6      const ImmutablePoint(this.x, this.y);
7  }

```

Los constructores constantes no siempre crean constantes. Pueden invocarse fuera de un `const` contexto. Para obtener más información, consulte la sección sobre el uso de constructores

5. Constructores de redireccionamiento

Un constructor puede redirigir a otro constructor de la misma clase. Un constructor que redirige tiene un cuerpo vacío. El constructor utiliza `this` en lugar del nombre de la clase después de dos puntos (`:`).

```
1 class Point {
2     double x, y;
3
4     // The main constructor for this class.
5     Point(this.x, this.y);
6
7     // Delegates to the main constructor.
8     Point.alongXAxis(double x) : this(x, 0);
9 }
```

2.3. Encapsulacion, Getters y Setters

En Dart, la encapsulación consiste en ocultar datos dentro de una biblioteca, protegiéndolos de factores externos. Esto ayuda a controlar el programa y evita que se vuelva demasiado complejo.[2]

La encapsulación se puede lograr mediante:

- Declarar las propiedades de la clase como privadas utilizando guion bajo (`_`).
- Proporcionar métodos públicos `getter` y `setter` para acceder y actualizar el valor de la propiedad privada.

Nota: Dart no admite palabras clave como `'public'`, `'private'` y `'protected'`. Dart utiliza `"_"` (guion bajo) para indicar que una propiedad o método es privado. Los métodos `getter` y `setter` se utilizan para acceder y actualizar el valor de una propiedad privada. Los métodos `getter` se utilizan para acceder al valor de una propiedad privada. Los métodos `setter` se utilizan para actualizar el valor de una propiedad privada.[2]

Ejemplo 1: En este ejemplo, crearemos una clase llamada **Employee**. Esta clase tendrá dos propiedades privadas: `_id` y `_name`. También crearemos dos métodos públicos, `getId()` y `getName()`, para acceder a las propiedades privadas. Asimismo, crearemos dos métodos públicos, `setId()` y `setName()`, para actualizar las propiedades privadas.[2]

```
1 class Employee {
2     // Private properties
3     int? _id;
4     String? _name;
5
6     // Getter method to access private property _id
7     int getId() {
8         return _id!;
9     }
10    // Getter method to access private property _name
11    String getName() {
12        return _name!;
13    }
14    // Setter method to update private property _id
```

```

15     void setId(int id) {
16         this._id = id;
17     }
18     // Setter method to update private property _name
19     void setName(String name) {
20         this._name = name;
21     }
22
23 }
24
25 void main() {
26     // Create an object of Employee class
27     Employee emp = new Employee();
28     // setting values to the object using setter
29     emp.setId(1);
30     emp.setName("John");
31
32     // Retrieve the values of the object using getter
33     print("Id: ${emp.getId()}");
34     print("Name: ${emp.getName()}");
35 }

```

En Dart, puedes controlar el acceso a las propiedades e implementar la encapsulación mediante el uso de propiedades de solo lectura. Para ello, añade la palabra clave ‘final’ antes de la declaración de la propiedad. De esta forma, solo podrás acceder a su valor, pero no modificarlo.[2]

```

1 class Student {
2     final _schoolname = "ABC School";
3
4     String getSchoolName() {
5         return _schoolname;
6     }
7 }
8
9 void main() {
10     var student = Student();
11     print(student.getSchoolName());
12     // This is not possible
13     //student._schoolname = "XYZ School";
14 }

```

Puedes crear métodos getter y setter usando las palabras clave **get** y **set**. En el siguiente ejemplo, hemos creado una clase llamada **Vehicle**. Esta clase tiene dos propiedades privadas: **_model** y **_year**. También hemos creado dos métodos getter y setter para cada propiedad. Estos métodos se llaman **model** y **year**, y se utilizan para acceder y actualizar el valor de las propiedades privadas.[2]

```

1 class Vehicle {
2     String _model;
3     int _year;

```



```

4
5 // Getter method
6 String get model => _model;
7
8 // Setter method
9 set model(String model) => _model = model;
10
11 // Getter method
12 int get year => _year;
13
14 // Setter method
15 set year(int year) => _year = year;
16 }
17
18 void main() {
19     var vehicle = Vehicle();
20     vehicle.model = "Toyota";
21     vehicle.year = 2019;
22     print(vehicle.model);
23     print(vehicle.year);
24 }

```

¿Por qué es importante la encapsulación?

1. Ocultación de datos : La encapsulación oculta los datos del exterior. Impide que el código fuera de la clase acceda a ellos. Esto se conoce como ocultación de datos.[2]
2. Facilidad de prueba : La encapsulación permite probar la clase de forma aislada. Esto permite probar la clase sin necesidad de probar el código fuera de ella.[2]
3. Flexibilidad : La encapsulación permite cambiar la implementación de la clase sin afectar al código que se encuentra fuera de la clase.[2]
4. Seguridad : La encapsulación permite restringir el acceso a los miembros de la clase. Esto permite limitar el acceso a los miembros de la clase desde el código externo a la biblioteca.[2]

2.4. Herencia

La herencia es la compartición de comportamiento entre dos clases. Permite definir una clase que extiende la funcionalidad de otra. La palabra clave ‘extend’ se utiliza para heredar de la clase padre.[2]

Cuando se utiliza la herencia, siempre se crea una relación ^{es} un/una ^{entre} la clase padre y la clase hija, como por ejemplo: Estudiante es una Persona , Camión es un Vehículo , Vaca es un Animal , etc.

Dart admite herencia simple, lo que significa que una clase solo puede heredar de una única clase. Dart no admite herencia múltiple, lo que significa que una clase no puede heredar de varias clases.[2]

```

1 class ParentClass {
2     // Parent class code
3 }
4
5 class ChildClass extends ParentClass {

```

```
6 // Child class code
7 }
```

En esta sintaxis, ParentClass es la superclase y ChildClass es la subclase. ChildClass hereda las propiedades y métodos de ParentClass .

Términos:

- Clase padre: La clase cuyas propiedades y métodos hereda otra clase se denomina clase padre. También se la conoce como clase base o superclase.[2]
- Clase hija: La clase que hereda las propiedades y métodos de otra clase se denomina clase hija. También se la conoce como clase derivada o subclase.[2]

Ejemplo 1: En este ejemplo, crearemos una clase Persona y luego crearemos una clase Estudiante que hereda las propiedades y métodos de la clase Persona

```
1 class Person {
2     // Properties
3     String? name;
4     int? age;
5
6     // Method
7     void display() {
8         print("Name: $name");
9         print("Age: $age");
10    }
11 }
12 // Here In student class , we are extending the
13 // properties and methods of the Person class
14 class Student extends Person {
15     // Fields
16     String? schoolName;
17     String? schoolAddress;
18
19     // Method
20     void displaySchoolInfo() {
21         print("School Name: $schoolName");
22         print("School Address: $schoolAddress");
23     }
24 }
25
26 void main() {
27     // Creating an object of the Student class
28     var student = Student();
29     student.name = "John";
30     student.age = 20;
31     student.schoolName = "ABC School";
32     student.schoolAddress = "New York";
33     student.display();
34     student.displaySchoolInfo();
35 }
```

Ventajas de la herencia:

- Promueve la reutilización del código y reduce el código redundante
- Ayuda a diseñar un programa de mejor manera.
- Simplifica y limpia el código, y ahorra tiempo y dinero en mantenimiento.
- Facilita la creación de bibliotecas de clases.
- Puede utilizarse para imponer una interfaz estándar a todas las clases hijas.

2.5. Polimorfismo

Poli significa muchos y morfo significa formas . El polimorfismo es la capacidad de un objeto para adoptar muchas formas. Como seres humanos, tenemos la capacidad de adoptar muchas formas. Podemos ser estudiantes, profesores, padres, amigos, etc. De manera similar, en la programación orientada a objetos, el polimorfismo es la capacidad de un objeto para adoptar muchas formas.[2]

La sobrescritura de métodos es una técnica que permite crear un método en la clase hija con el mismo nombre que el método en la clase padre. El método en la clase hija sobrescribe el método en la clase padre.[2]

```
1 class ParentClass{
2   void functionName(){
3   }
4 }
5 class ChildClass extends ParentClass{
6   @override
7   void functionName(){
8   }
9 }
```

Ejemplo 1: Polimorfismo mediante la sobrescritura de métodos en Dart

En el siguiente ejemplo, existe una clase llamada Animal con un método llamado eat() . El método eat() se redefine en la clase hija llamada Dog .

```
1 class Animal {
2   void eat() {
3     print("Animal is eating");
4   }
5 }
6
7 class Dog extends Animal {
8   @override
9   void eat() {
10    print("Dog is eating");
11  }
12 }
13
14 void main() {
15   Animal animal = Animal();
16   animal.eat();
17 }
```

```

17
18   Dog dog = Dog();
19   dog.eat();
20 }

```

Ventaja de usar polimorfismo:

1. Las subclases pueden anular el comportamiento de la clase padre.
2. Nos permite escribir código más flexible y reutilizable.

2.6. Clase abstracta

Las clases abstractas son clases que no se pueden inicializar. Se utilizan para definir el comportamiento de una clase que puede ser heredado por otras clases. Una clase abstracta se declara utilizando la palabra clave `abstract`.^[2]

```

1  abstract class ClassName {
2      //Body of abstract class
3
4      method1();
5      method2();
6  }

```

Un método abstracto es un método que se declara sin implementación. Se declara con un punto y coma (;) en lugar del cuerpo del método.^[2]

```

1  abstract class ClassName {
2      //Body of abstract class
3      method1();
4      method2();
5  }

```

Las subclases de una clase abstracta deben implementar todos los métodos abstractos de la clase abstracta. Se utiliza para lograr la abstracción en el lenguaje de programación Dart.^[2]

Ejemplo 1: En el siguiente ejemplo, hay una clase abstracta Vehículo con dos métodos abstractos `start()` y `stop()`. Las subclases `Auto` y `Bicicleta` implementan los métodos abstractos y los sobrescriben para imprimir el mensaje..^[2]

```

1  abstract class Vehicle {
2      // Abstract method
3      void start();
4      // Abstract method
5      void stop();
6  }
7
8  class Car extends Vehicle {
9      // Implementation of start()
10     @override
11     void start() {

```

```

12     print('Car started ');
13 }
14
15 // Implementation of stop()
16 @override
17 void stop() {
18     print('Car stopped ');
19 }
20 }
21
22 class Bike extends Vehicle {
23     // Implementation of start()
24     @override
25     void start() {
26         print('Bike started ');
27     }
28
29     // Implementation of stop()
30     @override
31     void stop() {
32         print('Bike stopped ');
33     }
34 }
35
36 void main() {
37     Car car = Car();
38     car.start();
39     car.stop();
40
41     Bike bike = Bike();
42     bike.start();
43     bike.stop();
44 }

```

Nota: La clase abstracta se utiliza para definir el comportamiento de una clase que puede ser heredada por otras clases. Se puede definir un método abstracto dentro de una clase abstracta.[2]

No se puede crear un objeto de una clase abstracta. Sin embargo, se puede definir un constructor en una clase abstracta. El constructor de una clase abstracta se invoca cuando se crea un objeto de una subclase.[2]

Ejemplo 2: En el siguiente ejemplo, hay una clase abstracta Bank con un constructor que toma dos parámetros: nombre y tasa. Hay un método abstracto interest(). Las subclases SBI e ICICI implementan el método abstracto y lo sobrescriben para imprimir la tasa de interés.

```

1 abstract class Bank {
2     String name;
3     double rate;
4
5     // Constructor
6     Bank(this.name, this.rate);

```

```

7
8 // Abstract method
9 void interest ();
10
11 //Non-Abstract method: It have an implementation
12 void display () {
13     print ('Bank Name: $name ');
14 }
15 }
16
17 class SBI extends Bank {
18     // Constructor
19     SBI(String name, double rate) : super(name, rate);
20
21     // Implementation of interest ()
22     @override
23     void interest () {
24         print ('The rate of interest of SBI is $rate ');
25     }
26 }
27
28 class ICICI extends Bank {
29     // Constructor
30     ICICI(String name, double rate) : super(name, rate);
31
32     // Implementation of interest ()
33     @override
34     void interest () {
35         print ('The rate of interest of ICICI is $rate ');
36     }
37 }
38
39 void main () {
40     SBI sbi = SBI ('SBI', 8.4);
41     ICICI icici = ICICI ('ICICI', 7.3);
42
43     sbi.interest ();
44     icici.interest ();
45     icici.display ();
46 }

```

2.7. Archivo y dart:io

La biblioteca dart:io proporciona API para gestionar archivos, directorios, procesos, sockets, Web-Sockets y clientes y servidores HTTP. [3]

En general, la biblioteca dart:io implementa y promueve una API asíncrona. Los métodos síncronos pueden bloquear fácilmente una aplicación, dificultando su escalabilidad. Por lo tanto, la mayoría de las operaciones devuelven resultados mediante objetos Future o Stream, un patrón común en plataformas de servidor modernas como Node.js.[3]

Los pocos métodos sincrónicos de la biblioteca `dart:io` están claramente marcados con el sufijo `Sync` en el nombre del método. No se tratan aquí los métodos sincrónicos.[3]

```
1 import 'dart:io';
```

A `File` contiene una ruta donde se pueden realizar operaciones. Puedes obtener el directorio padre del archivo usando `parent`, una propiedad heredada de `FileSystemEntity`.

Cree un nuevo `File` objeto con una ruta para acceder al archivo especificado en el sistema de archivos desde su programa.

```
1 var myFile = File('file.txt');
```

La `File` clase contiene métodos para manipular archivos y su contenido. Con los métodos de esta clase, puede abrir y cerrar archivos, leerlos y escribir en ellos, crearlos y eliminarlos, y comprobar su existencia.[3]

Al leer o escribir un archivo, puede utilizar secuencias (con `openRead`), operaciones de acceso aleatorio (con `open`) o métodos convenientes como `readAsString`. [3]

La mayoría de los métodos de esta clase se ejecutan en pares síncronos y asíncronos; por ejemplo, `readAsString` y `readAsStringSync`. A menos que tenga una razón específica para usar la versión síncrona de un método, prefiera la versión asíncrona para evitar bloquear su programa. Si `path` es un enlace simbólico, en lugar de un archivo, entonces los métodos de `File` operan en el destino final del enlace, excepto `delete` y `deleteSync`, que operan en el enlace.

- Leer desde un archivo: El siguiente ejemplo de código lee todo el contenido de un archivo como una cadena utilizando el método asíncrono `readAsString`

```
1 import 'dart:async';
2 import 'dart:io';
3
4 void main() {
5   File('file.txt').readAsString().then((String contents) {
6     print(contents);
7   });
8 }
```

Una forma más flexible y útil de leer un archivo es con un `Stream`. Abra el archivo con `openRead`, que devuelve un `stream` que proporciona los datos del archivo como fragmentos de bytes. Lea el `stream` para procesar el contenido del archivo cuando esté disponible. Puede usar varios transformadores sucesivamente para manipular el contenido del archivo al formato requerido o para prepararlo para la salida.[3]

Es posible que desee utilizar una secuencia para leer archivos grandes, para manipular los datos con transformadores o para lograr compatibilidad con otra API, como `WebSockets`. [?]

```
1 import 'dart:io';
2 import 'dart:convert';
3 import 'dart:async';
4
5 void main() async {
6   final file = File('file.txt');
7   Stream<String> lines = file.openRead()
```

```

8      .transform(utf8.decoder)      // Decode bytes to UTF-8.
9      .transform(LineSplitter());   // Convert stream to individual lines.
10     try {
11       await for (var line in lines) {
12         print('$line: ${line.length} characters ');
13       }
14       print('File is now closed. ');
15     } catch (e) {
16       print('Error: $e');
17     }
18   }

```

- Escribir en un archivo: Para escribir una cadena en un archivo, utilice el método `writeAsString`

```

1 import 'dart:io';
2
3 void main() async {
4   final filename = 'file.txt';
5   var file = await File(filename).writeAsString('some content');
6   // Do something with the file.
7 }

```

También puedes escribir en un archivo usando un `Stream`. Abre el archivo con `openWrite`, que devuelve un `IOSink` donde puedes escribir datos. Asegúrate de cerrar el `IOSink` con el método `IOSink.close`.

```

1 import 'dart:io';
2
3 void main() async {
4   var file = File('file.txt');
5   var sink = file.openWrite();
6   sink.write('FILE ACCESSED ${DateTime.now()}\n');
7   await sink.flush();
8
9   // Close the IOSink to free system resources.
10  await sink.close();
11 }

```

Para evitar bloqueos involuntarios del programa, varios métodos son asíncronos y devuelven un `Future`. Por ejemplo, el método `length`, que obtiene la longitud de un archivo, devuelve un `Future`. Espere a que el `future` obtenga el resultado cuando esté listo.

```

1 import 'dart:io';
2
3 void main() async {
4   final file = File('file.txt');
5
6   var length = await file.length();
7   print(length);
8 }

```


2.8. Manejo de errores

Su código Dart puede generar y capturar excepciones. Las excepciones son errores que indican que ocurrió algo inesperado. Si no se captura la excepción, el aislamiento que la generó se suspende y, por lo general, el aislamiento y su programa se cierran.[4]

A diferencia de Java, todas las excepciones de Dart son excepciones no comprobadas. Los métodos no declaran qué excepciones podrían lanzar, y no es necesario capturar ninguna.[4]

Dart proporciona tipos `Exception` y `Error`, así como numerosos subtipos predefinidos. Por supuesto, puede definir sus propias excepciones. Sin embargo, los programas Dart pueden lanzar cualquier objeto no nulo como excepción, no solo objetos de excepción y error.[4]

```
1 throw FormatException('Expected at least 1 section');
```

Capturar una excepción impide que se propague (a menos que se vuelva a lanzar). Capturar una excepción permite gestionarla:

```
1 try {
2   breedMoreLlamas();
3 } on OutOfLlamasException {
4   buyMoreLlamas();
5 }
```

Para gestionar código que puede generar más de un tipo de excepción, se pueden especificar varias cláusulas `catch`. La primera cláusula `catch` que coincida con el tipo del objeto generado gestiona la excepción. Si la cláusula `catch` no especifica un tipo, puede gestionar cualquier tipo de objeto generado: [4]

```
1 try {
2   breedMoreLlamas();
3 } on OutOfLlamasException {
4   // A specific exception
5   buyMoreLlamas();
6 } on Exception catch (e) {
7   // Anything else that is an exception
8   print('Unknown exception: $e');
9 } catch (e) {
10  // No specified type, handles all
11  print('Something really unknown: $e');
12 }
```

Como se muestra en el código anterior, puede usar uno de los dos `on`, `catch` o ambos. Úselo cuando necesite especificar el tipo de excepción. Úselo `catch` cuando su manejador de excepciones necesite el objeto de excepción.[4]

Puedes especificar uno o dos parámetros para `catch()`. El primero es la excepción lanzada y el segundo es el seguimiento de la pila (un `StackTrace` objeto).

2.9. Programación asíncrona: futuros, asíncrono, espera

Las operaciones asíncronas permiten que su programa complete su trabajo mientras espera que finalice otra operación. Estas son algunas operaciones asíncronas comunes:

Obteniendo datos a través de una red. Escribiendo en una base de datos. Leyendo datos de un archivo. Estos cálculos asíncronos suelen proporcionar su resultado como un `Future`, si el resultado tiene varias partes, como un `Stream`. Estos cálculos introducen asincronía en un programa. Para adaptar esta asincronía inicial, otras funciones Dart simples también deben volverse asíncronas.

Para interactuar con estos resultados asíncronos, puede usar las palabras clave `.async` y `await`. La mayoría de las funciones asíncronas son simplemente funciones asíncronas de Dart que dependen, posiblemente en el fondo, de un cálculo inherentemente asíncrono.

2.9.1. Future

Un futuro (con "f" minúscula) es una instancia de la clase `Future` (con "F" mayúscula). Un futuro representa el resultado de una operación asíncrona y puede tener dos estados: incompleto o completo.

Incompleto

Al llamar a una función asíncrona, esta devuelve un futuro incompleto. Este futuro espera a que la operación asíncrona de la función finalice o arroje un error.

Terminado

Si la operación asíncrona se realiza correctamente, el futuro se completa con un valor. De lo contrario, se completa con un error.

Un futuro de tipo `FutureT` se completa con un valor de tipo `T`. Por ejemplo, un futuro con tipo `FutureString` produce un valor de cadena. Si un futuro no produce un valor utilizable, entonces su tipo es `FutureVoid`.

Completando con un error Si la operación asíncrona realizada por la función falla por cualquier motivo, el futuro se completa con un error.

```
1 Future<void> fetchUserOrder() {
2   // Imagine that this function is fetching user info from another service or database.
3   return Future.delayed(const Duration(seconds: 2), () => print('Large Latte'));
4 }
5
6 void main() {
7   fetchUserOrder();
8   print('Fetching user order...');
9 }
```

2.9.2. Trabajar con futuros: `async` y `await`

Las palabras clave `.async` y `await` proporcionan una forma declarativa de definir funciones asíncronas y utilizar sus resultados. Recuerde estas dos pautas básicas al usar `.async` y `await`: [5] Para definir una función asíncrona, agregue `async` al cuerpo de la función: La palabra clave `await` sólo funciona en `async` funciones.[5] A continuación se muestra un ejemplo que convierte `main()` de una función sincrónica a una asíncrona.[5]

Primero, agregue la palabra clave `async` antes del cuerpo de la función:

```
1 void main() async { }
```

Si la función tiene un tipo de retorno declarado, actualízelo a FutureT, donde T es el tipo del valor que devuelve la función. Si la función no devuelve un valor explícitamente, el tipo de retorno es Futurevoid:

```
1 Future<void> main() async { }
```

Ejemplo: funciones sincrónicas

```
1 String createOrderMessage() {
2   var order = fetchUserOrder();
3   return 'Your order is: $order';
4 }
5
6 Future<String> fetchUserOrder() =>
7   // Imagine that this function is
8   // more complex and slow.
9   Future.delayed(const Duration(seconds: 2), () => 'Large Latte');
10
11 void main() {
12   print('Fetching user order...');
13   print(createOrderMessage());
14 }
```

Ejemplo: funciones asincrónicas

```
1 Future<String> createOrderMessage() async {
2   var order = await fetchUserOrder();
3   return 'Your order is: $order';
4 }
5
6 Future<String> fetchUserOrder() =>
7   // Imagine that this function is
8   // more complex and slow.
9   Future.delayed(const Duration(seconds: 2), () => 'Large Latte');
10
11 Future<void> main() async {
12   print('Fetching user order...');
13   print(await createOrderMessage());
14 }
```

2.10. Patrones de diseño

El desarrollo de software es un proceso complejo que requiere un enfoque estructurado y eficiente para lograr resultados óptimos. En este contexto, los patrones de diseño de software se han convertido en una herramienta invaluable para los desarrolladores, brindando soluciones probadas y comprobadas para problemas comunes de diseño. Al comprender y aplicar estos patrones, los profesionales del desarrollo pueden mejorar significativamente la eficiencia y mantenibilidad del código, evitando errores costosos y optimizando el tiempo de desarrollo [6].

Los patrones de diseño o *designpatterns*, son una solución general, reutilizable y aplicable a diferentes problemas de diseño de software. Se trata de plantillas que identifican problemas en el

sistema y proporcionan soluciones apropiadas a problemas generales a los que se han enfrentado los desarrolladores durante un largo periodo de tiempo, a través de prueba y error [6].

Estos patrones proporcionan un enfoque estructurado y reutilizable para resolver situaciones recurrentes en el desarrollo de software. Los patrones de diseño ayudan a los desarrolladores a comunicarse y compartir soluciones eficientes y efectivas que han demostrado ser exitosas en el pasado [6].

Cada patrón de diseño tiene un propósito específico y ofrece una solución a un problema de diseño común. Al utilizar un patrón de diseño, los desarrolladores pueden aprovechar la experiencia acumulada y evitar reinventar la rueda, lo que resulta en un software más eficiente, mantenible y escalable [6].

Es importante tener en cuenta que los patrones de diseño no son algoritmos o código listo para usar, sino más bien pautas y descripciones de soluciones de diseño. Los desarrolladores deben adaptar e implementar los patrones de diseño de acuerdo con los requisitos y características específicas de su proyecto de software [6].

Los patrones de diseño de software se basan en principios y prácticas de diseño que han evolucionado a lo largo del tiempo. A menudo, se describen utilizando terminología y notaciones específicas para facilitar su comprensión y aplicación [6].

2.10.1. Tipos de patrones de diseño

Patrones creacionales

Los patrones de creación proporcionan diversos mecanismos de creación de objetos, que aumentan la flexibilidad y la reutilización del código existente de una manera adecuada a la situación. Esto le da al programa más flexibilidad para decidir qué objetos deben crearse para un caso de uso dado [6]. Dentro de los patrones creacionales se encuentra Singleton, que es el que restringe la creación de instancias de una clase a un único objeto.

```

1  class BaseDeDatos {
2      //Instancia est tica privada: la nica que existir en memoria.
3  static final BaseDeDatos _instancia = BaseDeDatos._constructorPrivado();
4
5      //Constructor privado: impide usar 'new BaseDeDatos()' desde fuera.
6      BaseDeDatos._constructorPrivado();
7
8      //Devuelve siempre la misma instancia.
9      factory BaseDeDatos() {
10         return _instancia;
11     }
12
13     void conectar() {
14         print("Conectado a la BD usando la instancia nica .");
15     }
16 }
17
18 void main() {
19     //Al intentar crear dos objetos, ambos apuntan al mismo sitio.
20     BaseDeDatos conexion1 = BaseDeDatos();
21     BaseDeDatos conexion2 = BaseDeDatos();
22
23     //Resultado: true (Es el mismo objeto Singleton)
24     print(identical(conexion1, conexion2));
25 }

```

Patrones estructurales

Facilitan soluciones y est ndares eficientes con respecto a las composiciones de clase y las estructuras de objetos. El concepto de herencia se utiliza para componer interfaces y definir formas de componer objetos para obtener nuevas funcionalidades.

Patrones de comportamiento

El patr n de comportamiento se ocupa de la comunicaci n entre objetos de clase. Se utilizan para detectar la presencia de patrones de comunicaci n ya presentes y pueden manipular estos patrones. [6].

El patr n MVC implementado se clasifica en un nivel superior como Patr n Arquitect nico. A diferencia de los patrones de dise o (como el Singleton) que resuelven problemas espec ficos de instanciaci n o comunicaci n, el MVC organiza la estructura fundamental y las responsabilidades de todo el sistema de software.

Su funcionamiento del patr n MVC se demuestra en el siguiente ejemplo, donde se observa la independencia entre la clase que almacena los datos del estudiante (Modelo) y la clase que los imprime (Vista)

```

1 //Modelo
2 class Estudiante {
3     String nombre;
4     String matricula;
5
6     Estudiante(this.nombre, this.matricula);
7 }
8
9 //Vista
10 class EstudianteView {
11     void imprimirDetalles(String nombre, String matricula) {
12         print("Estudiante:");
13         print("Nombre: $nombre");
14         print("Matr cula: $matricula");
15     }
16 }
17
18 //controlador
19 class EstudianteController {
20     Estudiante modelo;
21     EstudianteView vista;
22
23     EstudianteController(this.modelo, this.vista);
24
25     // Modifica el modelo
26     void setNombre(String nombre) {
27         modelo.nombre = nombre;
28     }
29
30     // Actualiza la vista
31     void actualizarVista() {
32         vista.imprimirDetalles(modelo.nombre, modelo.matricula);
33     }
34 }
35
36 void main() {
37     // 1. Crear el modelo (datos)
38     Estudiante modelo = Estudiante("Juan P rez", "12345");
39
40     // 2. Crear la vista (interfaz)
41     EstudianteView vista = EstudianteView();
42
43     // 3. El controlador une ambos
44     EstudianteController controller = EstudianteController(modelo, vista);
45
46     // El controlador actualiza los datos y refresca la pantalla
47     controller.setNombre("Juan P.");
48     controller.actualizarVista();
49 }

```

3. Desarrollo

A continuación se pasará a explicar detalladamente cada parte fundamental de los temas en el siguiente orden:

3.1. Archivos

El programa proporcionado presenta un menú interactivo que permite al usuario realizar tres operaciones fundamentales sobre archivos de texto: crear y escribir, leer, y sobrescribir. Cada una de estas acciones está encapsulada en funciones específicas que reflejan buenas prácticas de modularidad y control de flujo.

- 1. Crear y escribir archivos: La función `crearYEscribirArchivo()` permite al usuario crear un archivo nuevo y guardar texto línea por línea. Este proceso refleja el concepto de persistencia de datos y buffer manual.

```
1 File archivo = File(nombreArchivo);
2 archivo.writeAsStringSync(lineas.join('\n'));
```

- Se crea una instancia de `File`, que representa un archivo en el sistema.
- `writeAsStringSync` escribe el contenido de forma sincrónica, garantizando que el archivo se guarde antes de continuar.
- El uso de `lineas.join('\n')` simula un buffer: se acumulan las líneas en memoria antes de escribirlas.

- 2. Leer archivos existentes: La función `_leerArchivoExistente()` permite recuperar el contenido de un archivo previamente creado. Aquí se aplica el concepto de extracción de datos persistentes.

```
1 if (!archivo.existsSync()) {
2     print('\nEl archivo no existe.\n');
3     return;
4 }
5
6 String contenido = archivo.readAsStringSync();
```

- Se verifica la existencia del archivo con `existsSync()`, lo que evita errores de lectura.
- `readAsStringSync()` carga todo el contenido en una sola operación, útil para archivos pequeños.

- 3. Sobrescribir archivos: La función `_sobrescribirArchivo()` introduce el concepto de modificación destructiva. Antes de sobrescribir, se solicita confirmación explícita del usuario.

```
1 if (confirmacion == null || confirmacion.toUpperCase() != 'SI') {
2     print('\nOperacion cancelada.\n');
3     return;
4 }
```

- Se protege la integridad del archivo solicitando confirmación.
- Si el usuario acepta, se recopila nuevo contenido y se usa `writeAsStringSync` para reemplazar el archivo.

- Conexión entre funciones: Todas las funciones comparten un patrón de interacción:

```
1 String? entrada = stdin.readLineSync();
```

- Se usa entrada estándar para interactuar con el usuario, lo que permite construir un flujo conversacional.
- Este patrón refuerza la idea de que el usuario es parte activa del proceso de persistencia.

El diagrama estático que representa el código Archivos es el siguiente:

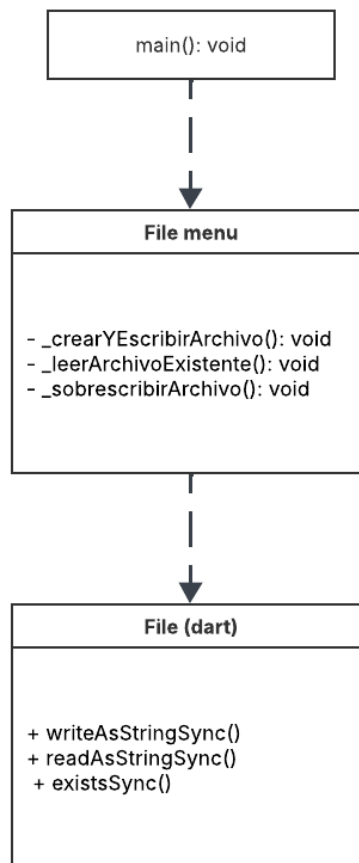


Figura 1: Diagrama estático / ARCHIVOS

El diagrama estático representa la estructura del programa en Dart que gestiona operaciones con archivos mediante un menú interactivo; la función principal `main()` actúa como punto de entrada y dirige el flujo hacia tres funciones privadas agrupadas conceptualmente en la clase `File menu`: `_crearYEscribirArchivo()`, `_leerArchivoExistente()` y `_sobrescribirArchivo()`, cada una encargada de una operación específica sobre archivos de texto; estas funciones interactúan directamente con la clase `File` de Dart, utilizando sus métodos `writeAsStringSync()`, `readAsStringSync()` y `existsSync()` para realizar escritura, lectura y verificación de archivos de forma sincrónica,

El diagrama dinámico que representa el código Archivos es el siguiente:

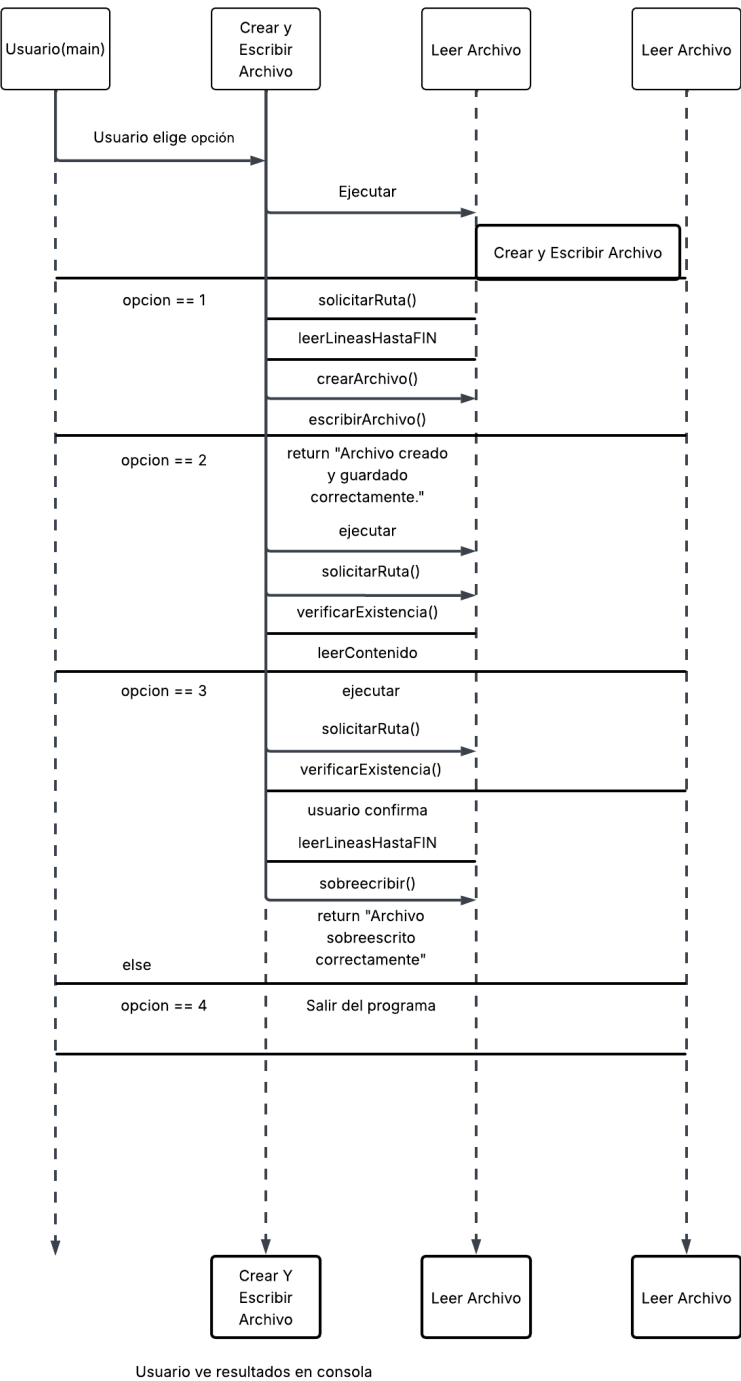


Figura 2: Diagrama dinámico / ARCHIVOS

El diagrama muestra cómo interactúa el usuario con tu programa, y qué métodos se ejecutan dependiendo de la opción que elija en el menú.

Hay 4 participantes:

- Usuario (main) - es quien elige la opción.
- crearYEscribirArchivo()
- leerArchivoExistente()
- sobrescribirArchivo()

Cada uno representa un flujo posible cuando el usuario selecciona del 1 al 4.

3.2. Hilos

Los programas proporcionados nos permiten comprender e ilustrar funciones básicas de la programación concurrente mediante hilos (threads) en Dart.

- Future.delayed: Esta función indica que una tarea se ejecutará después de un retraso específico, simulando operaciones asincrónicas.

:

```
1 Future.delayed(Duration(seconds: 2), () {  
2     print("Tarea asncrona completada");  
3 });
```

- Duration especifica el tiempo de espera antes de ejecutar la función.
 - La función anónima dentro de Future.delayed se ejecuta después del retraso, demostrando la naturaleza no bloqueante de las operaciones asincrónicas.
- await: La palabra clave await se utiliza para esperar la finalización de una operación asincrónica antes de continuar con la ejecución del código.

:

```
1 await Future.delayed(Duration(seconds: 2));  
2 print("Tarea completada con await");
```

- Aquí, el programa se pausa en la línea await hasta que la operación asincrónica (Future.delayed) se complete.
 - Esto permite escribir código que parece secuencial, pero que en realidad maneja operaciones asincrónicas de manera eficiente.
- receivePort y sendPort: Estos son mecanismos de comunicación entre hilos en Dart, permitiendo el intercambio de mensajes entre diferentes Isolates (hilos independientes).

:

```
1 void tarea(SendPort port){  
2     port.send("Hola desde otro isolate");  
3 }  
4  
5 void main() async {
```

```

6      final receivePort = ReceivePort();
7
8      await Isolate.spawn(tarea, receivePort.sendPort);
9
10     receivePort.listen((mensaje) {
11         print(mensaje);
12         receivePort.close();
13     });
14 }

```

- En este ejemplo, se crea un `ReceivePort` en el hilo principal para recibir mensajes.
 - Se inicia un nuevo `Isolate` (hilo) que ejecuta la función `tarea`, pasando su `SendPort` para enviar mensajes de vuelta al hilo principal.
 - El hilo principal “escucha” los mensajes entrantes y los procesa cuando llegan, permitiendo la comunicación entre hilos.
 - Al terminar de recibir mensajes, se cierra el `ReceivePort` para terminar el modo de escucha.
- `Isolate.spawn`: Esta función se utiliza para crear un nuevo `Isolate` (hilo) en Dart, permitiendo la ejecución concurrente de código y no detener el flujo principal.

:

```

1      import 'dart:isolate';
2      void sumaGrande(SendPort port) {
3          int total = 0;
4          for (int i = 0; i < 500000000; i++) {
5              total += i;
6          }
7          port.send(total);
8      }
9
10     void main() async {
11         final receive = ReceivePort();
12         print("Iniciando tarea pesada en hilo paralelo...");
13         await Isolate.spawn(sumaGrande, receive.sendPort);
14
15         print("Mientras tanto, sigo ejecutando en el hilo principal...");
16         receive.listen((resultado) {
17             print("Resultado: $resultado");
18             receive.close();
19         });
20     }

```

- En este ejemplo, se define una función `sumaGrande` que realiza una operación intensiva (sumar números hasta 500 millones) y envía el resultado de vuelta al hilo principal mediante un `SendPort`.
- En la función `main`, se crea un `ReceivePort` para recibir el resultado y se inicia un nuevo `Isolate` con `Isolate.spawn`, pasando la función `sumaGrande` y el `SendPort` del `ReceivePort`.

- Mientras el Isolate realiza la tarea pesada, el hilo principal continúa ejecutándose, con el objetivo de manejar tareas concurrentes sin bloquear la ejecución principal.
 - Finalmente, cuando el resultado está disponible, se recibe a través del ReceivePort y se imprime en la consola.
- Comunicación de dos direcciones entre Isolates: Aquí se muestra cómo establecer una comunicación bidireccional entre dos Isolates utilizando SendPort y ReceivePort.

:

```

1  import 'dart:isolate';
2
3  void worker(SendPort mainPort) {
4    final receive = ReceivePort();
5
6    mainPort.send(receive.sendPort);
7    receive.listen((msg) {
8      final respuesta = "Recibido en worker: $msg";
9      mainPort.send(respuesta);
10   });
11 }
12
13 void main() async {
14   final mainReceive = ReceivePort();
15
16   final isolate = await Isolate.spawn(worker, mainReceive.sendPort);
17
18   SendPort? workerSendPort;
19
20   mainReceive.listen((mensaje) {
21     if (mensaje is SendPort) {
22       workerSendPort = mensaje;
23       print("Main: recib el SendPort del worker.");
24
25       workerSendPort!.send("Mensaje desde main");
26     } else {
27       print("Main: respuesta del worker -> $mensaje");
28
29       mainReceive.close();
30       isolate.kill(priority: Isolate.immediate);
31     }
32   });
33 }

```

- En este ejemplo, se define una función worker que crea un ReceivePort para recibir mensajes del hilo principal y envía su SendPort de vuelta al hilo principal.
- En la función main, se crea un ReceivePort para recibir mensajes del worker y se inicia un nuevo Isolate con Isolate.spawn, pasando el SendPort del ReceivePort principal.
- El hilo principal escucha los mensajes entrantes; cuando recibe el SendPort del worker, lo almacena y envía un mensaje al worker.

- El worker procesa el mensaje recibido y envía una respuesta de vuelta al hilo principal.
 - Finalmente, el hilo principal imprime la respuesta del worker y cierra el ReceivePort, terminando la comunicación.
- Suma de números sin Isolates: Este ejemplo es usado para contrastarse con el ejemplo de suma con Isolates, mostrando cómo la ejecución secuencial ocupa más recursos y pone un estrés mayor en un núcleo singular.

:

```

1  void main() {
2  print("Inicio");
3
4  int total = 0;
5  for (int i = 0; i < 500000000; i++) {
6      total += i;
7  }
8
9  print("Resultado: $total");
10 print("Fin");
11 }

```

- En este ejemplo, la función main realiza una suma secuencial de números desde 0 hasta 500 millones.
- La ejecución se bloquea durante la operación intensiva, lo que puede hacer que la aplicación parezca no responder.
- Este enfoque es menos eficiente en términos de uso de recursos, ya que no aprovecha la capacidad de múltiples núcleos del procesador.

3.3. Patrones de diseño

Se describe la implementación de dos patrones de diseño en el contexto de la Ingeniería de Software, el patrón creacional tipo Singleton y el arquitectónico para un Modelo Vista Controlador (MVC) adaptado para un juego con temática de Pokemon.

3.3.1. Implementación de patrón en una gestión de impresoras

El problema maneja una sola cola de impresión compartida por múltiples usuarios mediante el patrón Singleton en su clase representativa *Impresora*. La estructura de la clase se diseñó para restringir la creación arbitraria de objetos, esto se logró mediante la definición de un constructor privado llamado *Impresora._interna()*, lo que impide que clases externas instancien la impresora directamente. En su lugar, se declaró una propiedad *static final Impresora _instancia* que almacena la única instancia de la clase, la cual es inicializada internamente.

Para permitir el acceso global a esta instancia única, se utilizó un constructor de tipo factory. Este constructor no crea un nuevo objeto cada vez que es llamado, sino que retorna siempre la referencia a la instancia estática previamente creada, en otras palabras, intercepta la llamada de creación y retorna siempre la *_instancia* que ya existe para no crear otro objeto. Además, la clase encapsula las estructuras de datos esenciales para el funcionamiento del sistema, protegiéndolas con el modificador de privacidad:

- Una cola *Queue < Documento > _colaImpresion* para manejar los documentos en orden FIFO.
- Una lista *List < Documento > _historial* para el registro de impresiones.

Se expusieron métodos públicos únicamente para agregar documentos, imprimir el siguiente en la cola y visualizar el estado actual, manteniendo ocultos los detalles internos de almacenamiento.

Posteriormente, hay que validar que el patrón funcione pues no deben existir múltiples impresoras, para ello se ingresaron las variables *impresoraA* e *impresoraB* en el *main*, la función *identical(impresoraA, impresoraB)* arroja un *true*, lo que demuestra que existe un espacio compartido entre las variables pues apuntan al mismo objeto Singleton. Finalmente, tres usuarios distintos (Alice, Bob y Carlos) enviaron documentos a través de las diferentes referencias de la impresora. Al imprimir, se observó que todos los documentos se centralizaron en una única cola de impresión, respetando el orden de llegada independientemente de la referencia utilizada para enviarlos.

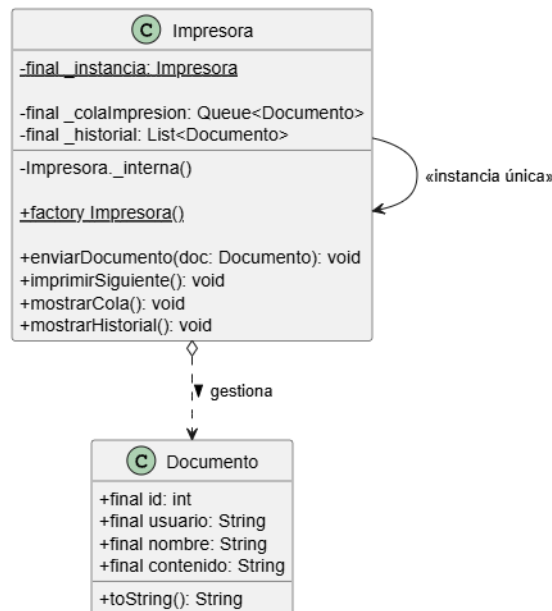


Figura 3: Diagrama de clases estático del Patrón Singleton aplicado al funcionamiento de una impresora

El diagrama de clases (Figura 3) ilustra la estructura interna de la clase *Impresora* diseñada para restringir su instanciación, esto lo logra mediante el atributo estático que almacena la única instancia de la clase. La flecha de asociación que sale de la clase y regresa a sí misma representa visualmente que *Impresora* es responsable de contener su propia instancia única.

El constructor aparece marcado como privado (-). Esto indica estructuralmente que ningún agente externo (como el *Main*) tiene permiso para utilizar el operador *new* de forma convencional para generar nuevos objetos, forzando el uso del punto de acceso controlado. Por otra parte, se expone un método público (+), definido como *factory*, que actúa como la única puerta de entrada para obtener la referencia de la impresora.

Las estructuras de datos como la *colaImpresion* y el historial se muestran como atributos privados, asegurando que el estado de la impresión solo pueda ser modificado a través de los métodos de negocio expuestos (*enviarDocumento*, *imprimirSiguiente*), protegiendo así la integridad de la fila única.

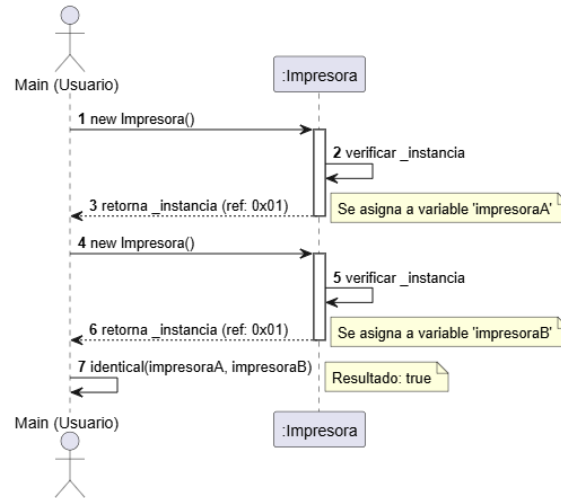


Figura 4: Diagrama dinámico del Patrón Singleton aplicado al funcionamiento de una impresora

Respecto al diagrama de secuencia, modela el comportamiento del sistema en tiempo de ejecución durante la solicitud de recursos, destacan entre las más concurrentes al usuario realizando dos llamadas consecutivas al constructor de la *Impresora*. Se evidencia que, aunque el usuario solicita una "nueva impresora" dos veces, el sistema no crea dos objetos distintos pues en la primera llamada, se retorna la referencia a la instancia estática *ref : 0x01*.

En la segunda llamada, el diagrama muestra que el objeto Singleton verifica la existencia de la instancia previa y retorna exactamente la misma referencia de memoria *ref : 0x01*. El flujo finaliza con una validación de identidad *identical* confirmando que ambas variables locales en el *Main* apuntan al mismo objeto físico en la memoria, cumpliendo así con el propósito fundamental del patrón: garantizar la unicidad del recurso compartido.

3.3.2. Juego de combate Pokémon

Se opta por el patrón MVC porque desacopla la lógica de combate con la interfaz del usuario, su capa del modelo define clases que representan las entidades del dominio (modelan los datos puros), encapsulando sus atributos y comportamientos básicos en la clase *Pokemon* con atributos como vida, velocidad y tipo, además de subclases específicas para tipos *Fuego* y *Hierba* tomados como ejemplos.

De igual manera, se modelaron los Ataques con su respectiva potencia y tipo, esta capa no tiene conocimiento alguno de cómo se muestran los datos ni de cómo se controla el flujo del juego. Se creó la clase abstracta *abstractclassCombateView* que define el contrato de métodos como *mostrarAtaque()* o *mostrarGanador()*. La implementación concreta, *ConsoleCombateView*, es la única parte del código que interactúa con la salida estándar *print*.

También se tiene una capa de controlador *CombateController*, es el cerebro, mantiene una

referencia a la vista mediante la propiedad final *CombateViewview*;, inyectada en su constructor, además, contiene el bucle principal del juego dentro del método *iniciarCombate()*, donde evalúa la lógica sobre quién es más rápido, cálculo de daño (lo duplica si es el caso) y dirige las llamadas a la vista para narrar el combate paso a paso.

Para comprobar su funcionamiento, se simuló un combate instanciando un *CombateController* y pasándole dos objetos del modelo (Charmander y Bulbasaur para este caso). Al ejecutar el programa, el controlador gestionó los turnos automáticamente basándose en el atributo velocidad del modelo, y la vista imprimió la secuencia de batalla en la consola, confirmando la separación de responsabilidades entre las tres capas. La consola mostró la narrativa del combate turno por turno, se verificó que cuando el personaje de tipo Fuego atacó al de tipo Hierba, el sistema calculó correctamente la ventaja de tipo (daño x2) y la Vista mostró el mensaje correspondiente. El ciclo finalizó automáticamente cuando la vida de uno de los objetos llegó a cero, declarando al ganador correctamente.

Para su diagrama estático:

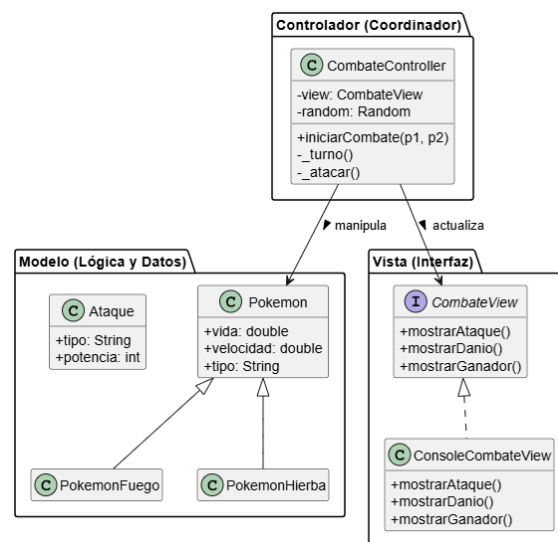


Figura 5: Diagrama de clases estático del Patrón MVC aplicado a un juego de Pokémon

Organiza los componentes del sistema en tres paquetes lógicos distintos; el primero es la independencia del modelo, el paquete *Modelo* (clases *Pokemon*, *Ataque*) aparece aislado de la *Vista*. Esto indica que las entidades de datos no tienen conocimiento de cómo se representan en pantalla, debido a que solo contienen atributos (vida, velocidad) y lógica pura de negocio. Posteriormente, para una abstracción de la vista, se muestra una interfaz *CombateView* que define un contrato claro para *mostrarAtaque* y *mostrarGanador*, mientras que, la clase *CombateController* depende de esta abstracción y no de la implementación concreta de *ConsoleCombateView*, lo que permitiría cambiar la consola por una interfaz gráfica en el futuro sin modificar el controlador.

Además, el controlador a través de la clase *CombateController* se sitúa como el eje central, teniendo asociaciones dirigidas hacia el *Modelo* para leer/escribir datos y hacia la *Vista* para enviar mensajes. Esto valida su rol de intermediario que gestiona la interacción entre los datos y el usuario.

Para el diagrama de secuencia, detalla el flujo de ejecución de un turno de combate, ilustrando la orquestación realizada por el Controlador:

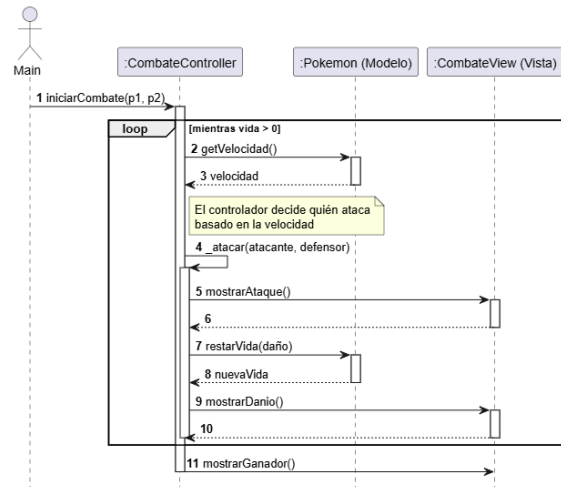


Figura 6: Diagrama de secuencia del patrón MVC

La secuencia comienza cuando el *Main* cede el control al llamar al método *iniciarCombate* del Controlador. A partir de este punto, el Controlador dirige el flujo. Antes de ejecutar cualquier acción visible, el Controlador consulta al modelo *getVelocidad* para determinar qué entidad debe actuar primero, demostrando que la lógica de decisión reside en el controlador y no en la vista, al ocurrir un ataque, el Controlador envía un mensaje al modelo *restarVida* para modificar su estado interno.

Inmediatamente después de que el modelo cambia o ocurre un evento, el Controlador invoca los métodos de la Vista *mostrarAtaque* y *mostrarDanio* para reflejar estos cambios al usuario. Finalmente el diagrama muestra un bucle cuya condición de terminación depende directamente del estado del modelo mientras la vida sea mayor que cero.

4. Resultados

4.1. Archivos

```
PS C:\P00_flutter\Archivos> dart archivos.dart
=====
          MENÚ DE ARCHIVOS
=====
1) Crear archivo .txt y escribir texto
2) Leer archivo existente
3) Sobrescribir archivo existente
4) Salir
Elige una opción: 1
Nombre del archivo a crear (ej: notas.txt): Ejemplo.txt

Escribe el texto que deseas guardar.
Para terminar, escribe SOLO: FIN
-----
Este ejemplo es lo mejor voy a reprobar calculo
FIN

Archivo creado y guardado correctamente.

=====
          MENÚ DE ARCHIVOS
=====
1) Crear archivo .txt y escribir texto
2) Leer archivo existente
3) Sobrescribir archivo existente
4) Salir
Elige una opción: 2
Ingresa el nombre o ruta del archivo a leer: Ejemplo.txt

===== CONTENIDO DEL ARCHIVO =====
Este ejemplo es lo mejor voy a reprobar calculo
=====
```

En la imagen anterior podemos observar el menú de la aplicación donde primero usamos la opción 1 para crear un archivo y escribir en este, para después usar la opción 2 para leer lo que escribimos confirmando que se escribió bien.

```

=====
          MENÚ DE ARCHIVOS
=====
1) Crear archivo .txt y escribir texto
2) Leer archivo existente
3) Sobrescribir archivo existente
4) Salir
Elige una opción: 3
Ingresa el nombre o ruta del archivo a sobrescribir: Ejemplo.txt

SE ENCONTRÓ EL ARCHIVO.
¿Deseas sobrescribirlo? Esto borrará todo su contenido.
Escribe "SI" para confirmar: SI

Escribe el nuevo contenido del archivo.
Cuando termines, escribe: FIN
-----
Parapapapa
FIN

Archivo sobrescrito correctamente.

=====
          MENÚ DE ARCHIVOS
=====
1) Crear archivo .txt y escribir texto
2) Leer archivo existente
3) Sobrescribir archivo existente
4) Salir
Elige una opción: 2
Ingresa el nombre o ruta del archivo a leer: Ejemplo.txt

===== CONTENIDO DEL ARCHIVO =====
Parapapapa
=====

=====
          MENÚ DE ARCHIVOS
=====
1) Crear archivo .txt y escribir texto
2) Leer archivo existente
3) Sobrescribir archivo existente
4) Salir
Elige una opción: 4
Saliendo del programa...
PS C:\POO_flutter\Archivos> 

```

En esta imagen podemos observar el uso de la opción 3 del menú, en la cual sobrescribimos

el archivo eliminando lo antes escrito, volvemos a usar la opción 2 para comprobar que se sobrescribió correctamente. Para finalizar usamos la opción 4 para salir de la aplicación

4.2. Hilos

```
PS C:\POO_flutter\Hilos> dart Ejemplo1.dart
Inicio
Fin inmediato (sin esperar)
Tarea asíncrona completada
PS C:\POO_flutter\Hilos>
```

Se muestra como es que el método future permite esperar otra tarea para que la otra tarea asíncrona pueda ocurrir. Se ve en la terminal la primer el fin de la primer tarea y luego de dos segundos el fin de la segunda.

```
PS C:\POO_flutter\Hilos> dart Ejemplo2.dart
Inicio
Tarea completada con await
Fin
PS C:\POO_flutter\Hilos>
```

En esta imagen se muestra como await afecta el tiempo y el modo de ejecución de un bloque de código.

```
PS C:\POO_flutter\Hilos> dart Ejemplo3.dart
Hola desde otro isolate
PS C:\POO_flutter\Hilos>
```

En la imagen se muestra que se mando un paso de mensajes de una tarea diferente mediante los puertos.

```
PS C:\POO_flutter\Hilos> dart Ejemplo4.dart
Iniciando tarea pesada en hilo paralelo...
Mientras tanto, sigo ejecutando en el hilo principal...
Resultado: 124999999750000000
PS C:\POO_flutter\Hilos>
```

Se muestra en la imagen como con el uso de isolate repartimos las tareas, mientras un hilo se encarga de las impresiones otro se encarga de hacer el cálculo.

```
PS C:\P00_flutter\Hilos> dart Ejemplo5.dart
Main: recibí el SendPort del worker.
Main: respuesta del worker -> Recibido en worker: Mensaje desde main
PS C:\P00_flutter\Hilos> █
```

En esta imagen podemos apreciar como mientras hay un puerto que recibe mensajes hay otro que los manda. Un hilo se encargará de mandar y otro de recibir. Se podría interpretar como una especie de chat primitivo entre hilos.

```
PS C:\P00_flutter\Hilos> dart Ejemplo6.dart
Inicio
Resultado: 124999999750000000
Fin
PS C:\P00_flutter\Hilos> █
```

Se muestra el uso de una suma serial sin repartir las tareas entre hilos.

4.3. Patronos

```
PS C:\POO_flutter\Patrones> dart main.dart
Nombre: Charmander
Nivel: 50
Tipo: Fuego
Vida: 231
Velocidad: 185
-----
Nombre: Bulbasaur
Nivel: 50
Tipo: Hierba
Vida: 288
Velocidad: 64
-----
===== COMBATE INICIADO =====
Charmander ataca a Bulbasaur con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
Bulbasaur ataca a Charmander con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
-- Siguiente turno --
Charmander ataca a Bulbasaur con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
Bulbasaur ataca a Charmander con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
-- Siguiente turno --
Charmander ataca a Bulbasaur con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
Bulbasaur ataca a Charmander con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
-- Siguiente turno --
Charmander ataca a Bulbasaur con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
Bulbasaur ataca a Charmander con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
-- Siguiente turno --
Charmander ataca a Bulbasaur con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
Bulbasaur ataca a Charmander con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
-- Siguiente turno --
Charmander ataca a Bulbasaur con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
Bulbasaur ataca a Charmander con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
¡Charmander se ha desmayado!
¡Charmander ha sido derrotado!, ¡Bulbasaur gana el combate!
===== COMBATE TERMINADO =====
PS C:\POO_flutter\Patrones> █
```

Se muestra en la imagen la simulación de un combate pokemon primero se muestran los pokemons y sus características, para después mostrar la simulación del combate entre estos, esta simulación se detiene cuando la vida de uno llega a cero haciendo vencedor al pokemon con vida.

```
PS C:\POO_flutter\Patrones> dart Singleton.dart
¿Es la misma instancia? true

Enviando a impresión: Documento #1 - "Reporte mensual" (Usuario: Alice)
Enviando a impresión: Documento #2 - "Contrato de servicio" (Usuario: Bob)
Enviando a impresión: Documento #3 - "Presentación proyecto X" (Usuario: Carlos)
=== Cola de impresión pendiente ===
Documento #1 - "Reporte mensual" (Usuario: Alice)
Documento #2 - "Contrato de servicio" (Usuario: Bob)
Documento #3 - "Presentación proyecto X" (Usuario: Carlos)
=====

--- Imprimiendo documento ---
Documento #1 - "Reporte mensual" (Usuario: Alice)
Contenido: Contenido del reporte mensual...
--- Fin de impresión ---

--- Imprimiendo documento ---
Documento #2 - "Contrato de servicio" (Usuario: Bob)
Contenido: Términos y condiciones del servicio...
--- Fin de impresión ---

=== Cola de impresión pendiente ===
Documento #3 - "Presentación proyecto X" (Usuario: Carlos)
=====

--- Imprimiendo documento ---
Documento #3 - "Presentación proyecto X" (Usuario: Carlos)
Contenido: Diapositivas del proyecto X...
--- Fin de impresión ---

No hay documentos en la cola de impresión.
=== Historial de documentos impresos ===
Documento #1 - "Reporte mensual" (Usuario: Alice)
Documento #2 - "Contrato de servicio" (Usuario: Bob)
Documento #3 - "Presentación proyecto X" (Usuario: Carlos)
=====

PS C:\POO_flutter\Patrones> █
```

Se muestra el funcionamiento de la app, que es una simulación de una impresora. Primero muestra los archivos que se mandaron a imprimir, así también muestra la cola de los archivos en impresión para después mostrar la simulación de la impresión de cada archivo y al final mostrar un historial de la impresora.

5. Conclusiones

La práctica desarrollada permitió integrar de manera efectiva tres componentes esenciales en el diseño de software: el manejo de archivos como medio de persistencia de datos, la programación concurrente mediante hilos para optimizar el rendimiento, y la aplicación de patrones de diseño como guía estructural para la construcción de sistemas robustos y escalables.

A través del análisis de ejemplos concretos y la elaboración de los diagramas UML estáticos y dinámicos, se logró visualizar con claridad tanto la arquitectura del sistema como su comportamiento en tiempo de ejecución. Esta representación gráfica facilitó la interpretación de los conceptos aplicados en el código, permitiendo identificar relaciones, dependencias y flujos de ejecución que no siempre son evidentes en la lectura directa del programa.

Más allá de la implementación técnica, el enfoque de la práctica se centró en la interpretación crítica de los conceptos, promoviendo una comprensión profunda de las decisiones de diseño y su impacto en la calidad del software. Este ejercicio fortaleció las habilidades de análisis y modelado, pero también fomentó una actitud reflexiva frente al desarrollo, donde cada componente se evalúa en función de su propósito, eficiencia y coherencia dentro del sistema.

Por ende, la práctica contribuyó a consolidar una visión integral del desarrollo de software, articulando teoría y aplicación en un entorno de aprendizaje guiado por ejemplos y representaciones formales.

Referencias

- [1] (2025) Dart language. Dart Developers. Consultado el 7 de noviembre de 2025. [Online]. Available: <https://dart.dev/language>
- [2] D. T. . D. Programming. (2025) Generic in dart. Consultado el 7 de noviembre de 2025. [Online]. Available: <https://dart-tutorial.com/object-oriented-programming/generics-in-dart/>
- [3] (2025) File class — dart:io. Flutter / Dart API Documentation. Consultado el 23 de noviembre de 2025. [Online]. Available: <https://api.flutter.dev/flutter/dart-io/File-class.html>
- [4] (2025) Manejo de errores en dart. Dart Developers. Consultado el 23 de noviembre de 2025. [Online]. Available: https://dart-dev.translate.goog/language/error-handling?_xt_rsl=en_xtrtl=es_xtrhl=es_xtrpto=tc
- [5] (2025) Async y await en dart. Dart Developers. Consultado el 23 de noviembre de 2025. [Online]. Available: https://dart-dev.translate.goog/libraries/async/async-await?_xt_rsl=en_xtrtl=es_xtrhl=es_xtrpto=tc
- [6] Profile, “Patrones de diseño de software. cómo resolver problemas de código comunes,” Blog de Profile, feb 2024, consultado el 28 de noviembre de 2025. [Online]. Available: <https://profile.es/blog/patrones-de-diseno-de-software/>